Sealed & Data classes – Data classes

Data classes are basically Java Pojo classes. In case where you implement a class which only **holds data,** use the Kotlin's **data class**.

When you define a **data class**, Kotlin will automatically generate the following methods for you

- getter & setter
- hashCode & equals
- toString
- copy
- componentX

```
data class User(val name: String,val age: Int)
```

Sealed & Data classes – Data classes

Java

```java
public class User {

   private String name;
   private int age;

   public User(String name, int age) {
      this.name = name;
      this.age = age;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getAge() {
      return age;
   }
   public void setAge(int age) {
      this.age = age;
   }
   @Override
   public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null || getClass() != o.getClass()) return false;

      User user = (User) o;

      if (age != user.age) return false;
      return name.equals(user.name);
   }
   @Override
   public int hashCode() {
      int result = name.hashCode();
      result = 31 * result + age;
      return result;
   }
   @Override
   public String toString() {
      return "User{" +
            "name='" + name + '\'' +
            ", age=" + age +
            '}';
   }
}
```

Kotlin

==

```kotlin
data class User(val name: String,val age: Int)
```

You spot the difference ?

Sealed & Data classes – Data classes

Class destructuring

Destructuring is the concecpt of treating objects as a set of seperate variables.

```
val (name, age) = user
val(_,age) = user
```

⇨

```
val name = person.component1()
val age = person.component2()
```

⇨

```
operator fun Person.component1() = getName()
operator fun Person.component2() = getAge()
```

Be aware that the reference is done by positional data

```
val users = listOf<User>()
for((name, age) in users){
    ...
}
```

Class copy

```
val user = User("Chris", 36)
val copy = user.copy(age = 21)
```

Sealed & Data classes – Sealed classes

Before we look into Sealed classes lets compare the enums of Java & Kotlin

Java‘s enum

```
public enum ViewState {
    ERROR,
    LAODING,
    DATA
}
```

Kotlin‘s enum

```
enum class ViewState{
    ERROR,
    DATA,
    LOADING
}
```

Both look very similar. Enums are already pretty consice in Java, so Kotlin doesn‘t save us any boiletplate code. Regarding funtionality, they are also very simple as there is not much you can with them...

```
fun handleViewState(viewState: ViewState): Unit = when(viewState){
    ViewState.ERROR -> { /* set error state in the view */ }
    ViewState.DATA ->  { /* hide loading state and process data ? */ }
    ViewState.LOADING -> { /* show loading indicator */ }
}
```

Sealed & Data classes – Sealed classes

So in essence, if you want to use enums for something simple that doesn‘t need to hold any data or different types of data per value, then they are fine.

But if we need something more specific we need something else.

And this is where Kotlin‘s **Sealed classes** come into play. (*Sealed classes are comparable to Swift‘s enum*)

- *Sealed classes are used for representing restricted class hierarchies.*
- *They are, in a sense, an extension of enum classes.*
- *A subclass of a sealed class can have multiple instances which can contain state.*

Sealed & Data classes – Sealed classes

*In short, **Sealed classes** are good when returning different but related results.*

```
sealed class ViewState {
    object Error: ViewState()
    object Loading: ViewState()
    data class Data(val someData: SomeData) : ViewState()
}
```

```
fun handleViewState(viewState: ViewState): Unit = when(viewState){
    is ViewState.Error -> { /* show error screen */ }
    is ViewState.Loading -> { /* show loading indicator */ }
    is ViewState.Data -> { handleData(viewState.someData)}
}
```

End of section: Sealed Classes
Any questions ?